



Proceedings of the Third International Workshop on Sustainable Ultrascale Computing Systems (NESUS 2016) Sofia, Bulgaria

Jesus Carretero, Javier Garcia Blas, Svetozar Margenov
(Editors)

October, 6-7, 2016

Jarus, M., Oleksiak, A., Narsisian, W. & Astsatryan, H. (2016). Energy-efficient Assignment of Applications to Servers by Taking into Account the Influence of Processes on Each Other. En *Proceedings of the Third International Workshop on Sustainable Ultrascale Computing Systems (NESUS 2016)* Sofia, Bulgaria (pp. 79-84). Madrid: Universidad Carlos III de Madrid. Computer Architecture, Communications, and Systems Group (ARCOS).

Energy-efficient Assignment of Applications to Servers by Taking into Account the Influence of Processes on Each Other

Mateusz Jarus[†], Ariel Oleksiak^{†‡}, Wahi Narsisian*, and Hrachya Astsatryan*

[†]Poznań Supercomputing and Networking Center, ul. Jana Pawła II 10, 61-139 Poznań

[‡]Poznań University of Technology, Pl. Marii Skłodowskiej-Curie 5, 60-965 Poznań

*Institute for Informatics and Automation Problems of the National Academy of Sciences of the Republic of Armenia, P. Sevak 1, Yerevan 0014, Armenia

Abstract

The power consumption of data centers is becoming a crucial challenge in the context of the steadily increasing demand for computation. In this regard finding a way to improve energy efficiency of running applications in data centers is becoming a crucial trend. One method to improve the processor utilization is the consolidation of applications on physical servers. It is possible to run multiple jobs in parallel on the same machine, especially when their requirements regarding computation are smaller than the maximum processor performance. It reduces the number of servers in the data center required to handle multiple requests and therefore leads to energy usage reductions. In this paper, we introduce a realistic model of applications with deadlines executed in parallel on a server and competing for the shared resources and present an energy-aware algorithm which may be used to minimize the overall energy consumption of the servers.

Keywords Data centers, Energy efficiency, Processor utilization, Applications scheduling

I. INTRODUCTION

Data centers are under pressure to transform their infrastructure to reduce energy cost, increase reliability and efficiency. Increasing the data volumes and network traffic in data centers is a worldwide trend. At the same time, the number of applications running in these data centers is becoming bigger and bigger over time. The types of the executed applications differ and include databases, file servers, middleware and various others. The difference between such data centers and typical HPC supercomputers is that it is natural in such places to co-locate dozens of tasks on a single physical node. It is a method for improving resource utilization. The relocation of the applications on servers is playing an important role to decrease the number of physical servers in data centers and to reduce the energy consumption.

In the case of data centers particularly important is the Service Layer Agreement which needs to be fulfilled. In our model, it is introduced in the form of deadlines for the tasks.

In this paper, we create a realistic model of applications with deadlines executed in parallel on a server, which compete for the shared resources, such as memory or disk. We explain the observations from the experiments that create the basis for the model.

We describe how the processor time quantum is shared between the applications and how their performance degrades through the use of the shared resources. We also present a Branch and Bound algorithm which may be used to minimize the overall energy consumption of the servers.

The remainder of this paper is divided into the following sections: Section 2 presents related work; Section 3 describes the model; Section 4 shows the performed experiments and their results; Section 5 concludes the paper.

II. RELATED WORK

As virtualization [1] has become the most widespread used technology in modern data centers, and due to the advances in virtualization technologies it is much easier to manage the allocation of tasks to the available resources. The live migration technique allows moving a running virtual machine from one physical server to another with no impact on virtual machine availability. Increasingly popular becomes the Docker platform, which allows starting up its containers even ten times faster than a standard virtual machine. The management of tasks is therefore very fast and efficient. However, the allocation of tasks to servers to maximize the utilization of resources

remains a challenge.

In [2] the authors illustrate the workload sensitivity to the machine on which it executed and the type of co-running applications. They analyzed co-running different applications on various processors and proved that it resulted in various levels of performance degradation of these jobs. The authors observed significant performance variability from the heterogeneity of the datacentre and from the co-allocation of applications. It is, therefore visible that in order to efficiently utilize available resources it is required to take into account the type of applications that are executed in parallel.

Multiple researchers have aimed at creating an algorithm to increase the utilization of machines in datacentres. In [3] the authors propose a Bubble-Up characterization methodology that enables the accurate prediction of the performance degradation that results from the contention for shared resources in the memory subsystem. Using this methodology they can improve the utilization of a 500-machines cluster by 50% to 90%.

In [4] the authors propose a performance model that considers the interferences in the shared last-level cache and memory bus. They also present a virtual machines consolidation method which is based on their interference model.

In [5] the authors propose a new resource management model for the collocation of different tasks that share a single physical machine. The model uses two parameters of a task – its size and its type – to characterize how a task influences the performance of other tasks allocated on the same machine.

However, all of the above methods are simplified. They take into account only one parameter of the application (such as memory accesses) or model the interference between applications by using one artificial parameter specified by the user. Experiments on real hardware prove that the dependencies between jobs are more complicated.

III. MODEL OF TASKS EXECUTED IN PARALLEL ON A SINGLE MACHINE

We propose a mathematical model that simulates the complex dependencies between co-running applications and hardware. It is based on the observation that each of the executed benchmarks affects the underlying hardware by utilizing its resources (processor, memory, hard drive, etc.). The load exerted on these subcomponents influences in turn other co-running applications – their performance degrades due to the need to compete for shared resources. The model does not try to simulate the interactions between applications per se, but rather captures the relationships between applications that appear when sharing the available resources.

In the model both the processor performance and the application, size is represented as Instructions Per Second (IPS). When all of the applications exert load that is equal to or smaller than 100%, the server has enough performance to efficiently execute all of

them. The situation becomes more complicated when the total requirements from applications are higher, for example, if each of the two applications requires 60% of the CPU load. In such case, they exceed the maximum processor performance. It is possible to execute them sequentially with the expected performance. They may also run in parallel but slower due to: a) the competition for shared resources, b) not satisfied CPU performance requirements. In both cases, the overall energy consumption and the duration of the execution may be analyzed.

To explain this mechanism in more detail, consider an application X that executes on a given server in T_1 seconds and exerts the L1 load on the CPU. Another application with L2 load on the CPU may be executed in parallel, where $L_1 + L_2 \leq 100\%$. The execution time of X will increase slightly due to the interference effect, as they will compete for shared resources, such as memory or disk. Starting additional applications will further extend the execution time of application X. As long as the aggregated load from all applications will be smaller than 100%, the performance degradation of application X will only result from the increasing load on the shared resources. However, when CPU load exceeds 100%, another factor of performance degradation becomes visible. The execution of the application is affected by periods of inactivity when it needs to wait for the processor time quantum.

We have performed a few experiments on Intel Core i5 6200U with three different applications, each exerting different load on the CPU: *pi* (30%), *siege* (50%) and *openssl* (65%). We tested the execution time of *pi* application while running additional applications in parallel. When the aggregated load was lower than 100%, the execution of *Pi* increased slightly. However, it increased significantly more after exceeding 100% CPU load, when all applications were executed in parallel.

The situation changes when the power consumption is considered. Starting additional applications increases the power consumption of the processor proportionally to the load that they make.

Similar experiment was performed with the same three applications, but this time to calculate the power consumption of the processor. The CPU power increased only until the CPU load was below 100%. After this point it stabilized. Since it is not possible to exceed the maximum processor speed, after reaching the point of 100% CPU load the power consumption did not change. However, the execution time of the applications increased significantly, having an impact on the whole energy consumption.

When realistic energy efficient job scheduling is considered, two challenges need therefore to be analyzed.

- the interference of applications on each other when they compete for shared resources,
- the calculation of the execution time of applications when their aggregated performance requirements exceed maximum processor performance.

III.1 Interference of applications

Our model is based on the observation that the applications do not affect directly each other but rather influence the underlying hardware, which in turn has an impact on the other executed applications. For example, accessing the memory by one application may cause a delay in accesses by another application.

In the model for each application different parameters regarding hardware may be specified, such as the number of memory accesses or disk usage. The more parameters are defined, the more accurate the results, but at the same time, the more data needs to be collected to run the experiments. For each application, there also needs to be defined a function of execution slowdown due to the aggregated load of a given subcomponent. It may be calculated using the Bubble-Up methodology, presented in [3]. It enables the accurate prediction of the performance degradation using a tunable amount of “pressure” to the subcomponent – memory in the case of this paper. “Bubble” is an artificial benchmark which is only used to stress the server memory. For different values of this pressure, the performance degradation of the original application is analyzed.

III.2 The extension of the execution time due to higher processor performance requirements

The model is based on the fact that the processor time quantum is consistently shared between the executed applications. This situation is presented in Figure 1. In this example, the maximum processor performance is 10 IPS and is named here “an execution window”. This run window is moved down in each second and shared between neighboring applications. Linux Completely Fair Scheduler is based on the same assumption that each application receives a fair amount of time quantum – according to its needs.

Figure 2 presents the new speed of execution of an application when the aggregated requirements of all applications are higher than the maximum processor performance. The size of the execution window is equal to the maximum processor performance – 10 IPS in this example. For the sake of clarity the applications are analyzed for a time which is equal to the time window – though the calculations are general and independent of the length of execution of any application. Variable x represents the exceeded processor performance. In this example there are four applications, x is calculated as $x = (s_1 + s_2 + s_3 + s_4) - perf$, where $perf$ is the maximum processor performance and s_1, s_2, s_3 and s_4 are the execution speeds or the CPU loads exerted by the four consecutive applications. New speed of any application may be calculated as $IPS_{new} = \frac{s \cdot c}{c} - \frac{s \cdot x}{c}$. For example, the original speed of the second application in Figure 2 was 4 IPS, while when running with three other applications in parallel it slows down to $\frac{40}{13}$ IPS $\approx 3,08$ IPS (interference effect due to the competition for shared resources is not considered in these calculations yet).

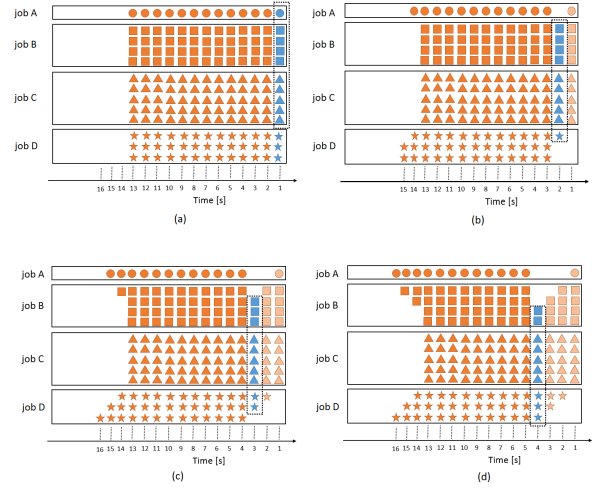


Figure 1: Model of the execution of the applications that exceed 100% of the processor load

IV. THE ENERGY-AWARE JOB SCHEDULING ALGORITHM

The input parameters for the algorithm are:

- the maximum processor performance (in IPS),
- the maximum power consumption of the processor,
- the number of all instructions for each application,
- the initial requirements for each application regarding processor performance (in IPS),
- the requirements for each application regarding its hardware usage (such as memory or disk usage),
- the function of performance degradation for each application due to the load exerted on different subcomponents of the server (such as memory or disk),
- the deadline for each application.

To calculate the optimal solution for a given processor and a number of various applications we implemented a Branch and Bound algorithm. It analyzes all correct instances of the problem. It starts with an array of a size $N \times N$, where N is the number of applications. Each analyzed job may be allocated to one of the $N \times N$ cells in the array. Columns represent the sequential execution of applications, while rows allow them to run in parallel. More generally – X axis represents passing time, while Y axis is the load of the CPU. An example instance of the problem presented in Figure 3 a). All of the jobs are allocated to the first column. Therefore all of them

should be executed at time 0 in parallel. Figure 3 b) shows their final execution on the processor. It is important to underline here that the array in Figure 3 a) does not take into account the length of the execution of any job and its requirements regarding processor performance. At this stage these values are not calculated, only their relative position against each other considered here.

Figure 4 a) presents another example of scheduling the tasks. In this case, there is a blank space between an orange and a green task. Figure 4 b) shows how these applications will be executed on the server. It represents a situation where the green task should not be executed in parallel with the blue task.

Please note that the position of the green task on the Y axis has no meaning, since there is no other job running in parallel. In this case only the height of the green task is significant as it represents the CPU load. In Figure 4 b) the green task may be therefore depicted at the same level as the blue task.

Please also note that if the green application would be scheduled in the same last column but in the lower row (the same row as the blue task), this allocation would not be correct. It would represent a situation in which there should be a delay of execution between the blue and the green task. However, since the orange application is shorter than the blue one, there is no other application that could separate them. Artificial delays of any length are not considered by the algorithm since they are useless. They do not improve the energy consumption and do not prevent from exceeding the deadlines. Such a schedule would be correct if the orange application would be longer than the blue one. At this stage this information is not available yet – the correctness of the instance validated at a later stage.

The algorithm creates all possible instances of the problem using a Branch and Bound technique. A few different instances of the problem are presented in Figure 5.

For each instance of the problem in the first step the algorithm calculates the time when each application finishes its execution. An example is presented in Figure 6. Vertical borders that represent these times create different phases of execution. If the length of the execution of each application is different, there are always as many stages as the number of applications, no matter what is their relative order. Please note that in each phase the same application might have a different speed of execution. In this example the maximum processor performance is not exceeded, therefore it does not contribute to a slowdown of any application. However, if that would be the case here, the green application in phase 2 would have a higher speed (higher height in the figure) because it would no longer share the processor time quantum with the blue application. For this reason, every phase needs to be analyzed separately.

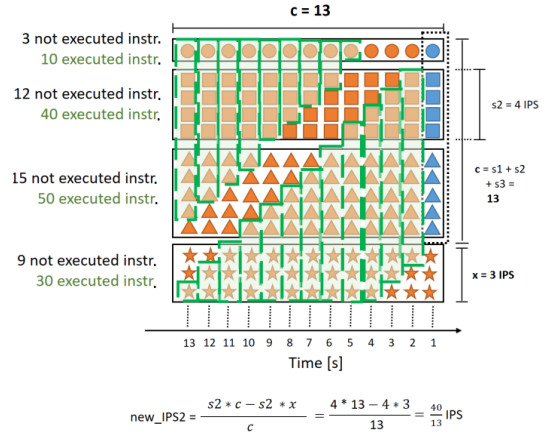


Figure 2: New speed of execution of an application after exceeding maximum processor performance

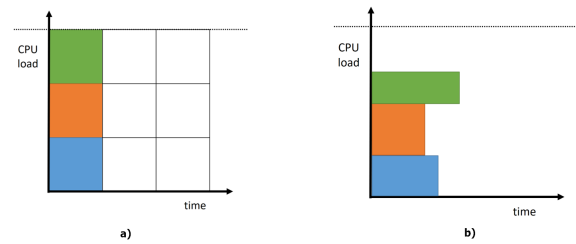


Figure 3: a) An example scheduling of the tasks and b) their final execution on the processor.

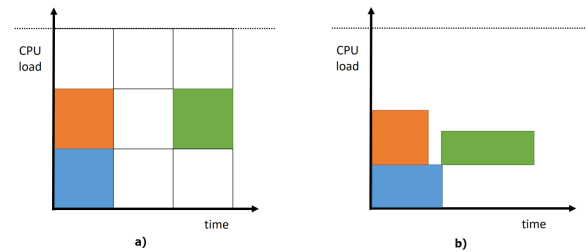


Figure 4: a) Another example of scheduling the tasks with a delay between an orange and a green task and b) their final execution on the processor.

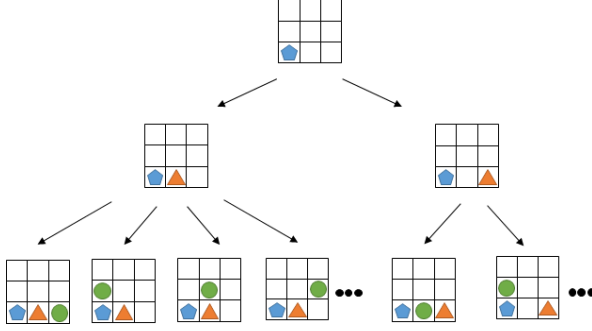


Figure 5: A few different instances of the problem created by the Branch and Bound algorithm.

In the second step for each row a bidirectional list is created (see Figure 6 c). Each item on these lists represents either a given job or an empty space between them. Each of these elements will hold the time when its phase finishes.

The algorithm iterates over every phase. For each list it saves the pointer to the currently analyzed item. For each phase it does two rounds – in the first one it analyzes items that are jobs, in the second round it analyzes blank items.

It starts with the first item on every list. If the first item on a given list represents a job (in our example on both lists the first items are jobs), it saves in it the execution time of this job, which is calculated as $time = instructions / speed$.

This value is added to the list *borders*, which holds information about the times of consecutive borders between phases. It moves the pointer of the currently analyzed item to the next one. While the next item on the given list is also a job, it repeats the same procedure – it calculates the time of the execution of this job. It adds to it the time of the previous item on the list and saves this value inside the currently analyzed item. It also adds it to the list *borders*. If the next item on a given list is a blank space, the algorithm moves to the next list and repeats the same procedure.

When all first jobs on each list are analyzed, the algorithm moves to the second phase – it examines blank spaces for each list. The algorithm checks whether the first item on the list *borders* is higher than the value saved for the previous item on the analyzed list. If yes, it saves it inside the item and moves the pointer to the currently analyzed item to the next one. If not, it leaves the item untouched. When all blank spaces in this phase for every list are checked, it removes the first item on the *borders* list.

The algorithm then moves to the next phase and repeats the whole procedure until all items on all lists are checked.

This step calculates the execution times for each job and the phases in which they are run. For instance, in the analyzed example, it shows that the green application is executed in phase 1 and 2, the blue one only in phase 1, while the orange one only in phase 3.

It also shows which applications are run in parallel with others in every phase.

Since the allocations to different stages are now known, the algorithm may calculate for each phase the processor load and the aggregated loads exerted on the subcomponents, such as memory. For example, in phase 1 it sums up the memory requirements of the green and the blue application. Then it analyzes the speed degradation for each of them under this aggregated memory load. Based on that information it updates the time of each phase, already saved in the previously mentioned lists. In the next step, it analyzes the new speed of each application according to the calculations presented in section III.2. Based on that information it again updates the time of each phase, already saved in the previously mentioned lists.

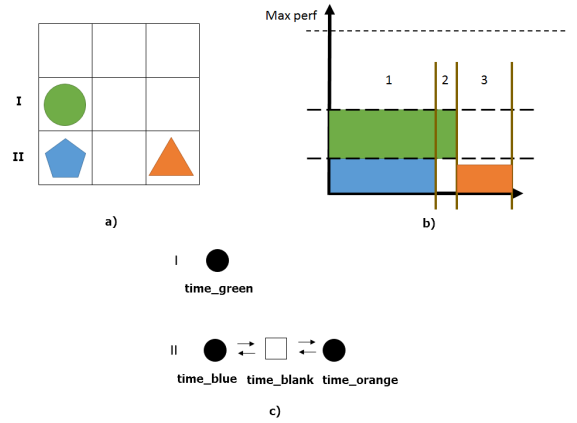


Figure 6: a) An example instance of the problem, b) marked the end of execution for each application (borders between phases) and c) the lists for each row that present the dependencies between tasks

The final result is the time of the end of every phase. It allows the algorithm to calculate the whole time required to run all of the applications. It also verifies the deadlines – if they are exceeded, the solution is treated as unacceptable. The energy consumption may be calculated as the time multiplied by the power (both of these values are known).

Since this is a Branch and Bound algorithm, the currently analyzed solution is compared to the previously saved. If it is better than the previous one, it will consider the best option. Finally, the algorithm returns the best instance from all analyzed.

V. EXPERIMENT

To test the algorithm we have started it with five applications. The applications compete for one shared resource, which is a memory.

The number of instructions, initial CPU and memory requirements and deadlines for each application is specified in Table 1. Applications three and four have high memory requirements. The first and the last applications have many instructions to execute (at the same time their execution would be the longest without taking into account the interference effects). All of them have specified deadlines.

Table 2 presents speed degradation of all applications in function of the memory load. For example, application one slows down by 5% when the memory load is 10 (e.g. Mb/s).

Name	Instructions	CPU	Memory	Deadline
one	1000	40	10	50
two	200	15	1	40
three	400	45	55	30
four	400	60	55	12
five	1000	30	10	40

Table 1: Parameters of five applications used to test the algorithm

	Application slowdown				
Memory	one	two	three	four	five
10	5,00%	5,00%	10,0%	15,0%	5,00%
20	6,00%	6,00%	18,0%	20,0%	6,00%
30	7,00%	6,30%	21,0%	25,0%	7,00%
40	7,30%	6,80%	23,0%	30,0%	7,30%
50	7,60%	7,10%	30,0%	35,0%	7,60%
60	8,00%	7,20%	35,0%	40,0%	8,00%
70	8,10%	7,20%	38,0%	45,0%	8,10%

Table 2: Speed degradation of applications in function of the memory load

Figure 7 presents the most energy-efficient scheduling for the proposed parameters. The time of calculations is 47,48 seconds and the energy consumed is 1751 Ws. In this solution, no deadlines are exceeded. It is also visible that the maximum processor performance is not exceeded in any phase. All of the applications are executed with the initially required speed.

VI. CONCLUSION

In this paper, we presented a model of applications with deadlines executed in parallel on a server, which compete for the shared resources, such as memory or disk. This model realistically represents the real execution of applications on physical servers, taking into account their speed, hardware requirements and performance degradation due to loaded subcomponents of the server. We presented a

Branch and Bound algorithm to calculate the most energy-efficient scheduling of jobs. The algorithm was verified by five applications with specified hardware requirements and deadlines.

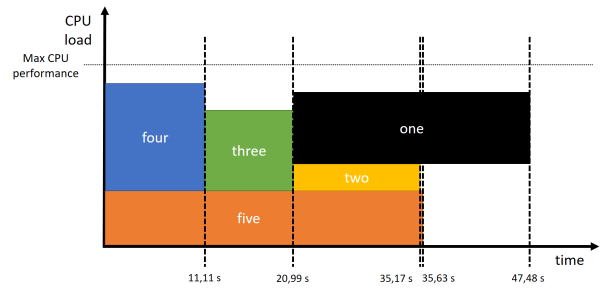


Figure 7: The most energy-efficient scheduling for the selected applications.

Acknowledgment

This work is partially supported by EU under the COST Program Action 1305: Network for Sustainable Ultrascale Computing (NESUS). The research presented in this paper is partially funded by a grant from Polish National Science Center under award number 2013/08/A/ST6/00296. This research was supported by the EU Seventh Framework Programme FP7/2007–2013 under grant agreement no. FP7-ICT-2013-10 (609757).

REFERENCES

- [1] M. Wang, X. Meng, L. Zhang, "Consolidating virtual machines with dynamic bandwidth demand in data centers" in: IEEE INFOCOM 2011 Proceedings, Shanghai, China, June 2011, pp. 71-75.
- [2] J. Mars, L. Tang, R. Hundt, "Heterogeneity in 'Homogeneous' Warehouse-Scale Computers: A Performance Opportunity", in: IEEE Computer Architecture Letters, vol. 10, No. 2, pp. 29-32, 2011.
- [3] J. Mars, L. Tang, K. Skadron, M. L. Soffa, R. Hundt, "Increasing Utilization in Modern Warehouse-Scale Computers Using Bubble-Up", in: IEEE Micro, vol. 32, pp. 88-99, 2012.
- [4] S. Kim, H. Eom, H. Y. Yeom, "Virtual machine consolidation based on interference modeling", in: The Journal of Supercomputing, vol. 66, pp. 1489-1506, 2013.
- [5] F. Pascual, K. Rzaqca, "Partition with side effects", in: IEEE 22nd International Conference on High Performance Computing, vol. 66, pp. 1489-1506, 2013.